Applied Mathematical Sciences, Vol. 7, 2013, no. 30, 1447 - 1476
HIKARI Ltd, www.m-hikari.com

Accelerating Kernel Density Estimation on the GPU Using the CUDA Framework

Panagiotis D. Michailidis

Department of Balkan Studies, University of Western Macedonia
3rd Km Florinas-Nikis National Road, Florina, 53100, Greece
pmichailidis@uowm.gr

Konstantinos G. Margaritis

Department of Applied Informatics, University of Macedonia
156 Egnatia str., Thessaloniki, 54006, Greece
kmarg@uom.gr

Abstract

The main problem of the kernel density estimation methods is the huge computational requirements, especially for large data sets. One way for accelerating these methods is to use the parallel processing. Recent advances in parallel processing have focused on the use Graphics Processing Units (GPUs) using Compute Unified Device Architecture (CUDA) programming model. In this work we discuss a naive and two optimised CUDA algorithms for the two kernel estimation methods: univariate and multivariate. These optimised algorithms are based on the use of shared memory tiles and loop unrolling techniques. We also present exploratory experimental results of the proposed CUDA algorithms according to the several values of parameters such as number of threads per block, tile size, loop unroll level, number of variables and data (sample) size. The experimental results show significant performance gains of all proposed CUDA algorithms over serial CPU version and small performance speed-ups of the two optimised CUDA algorithms over naive GPU algorithms. Finally, based on extended performance results are obtained general conclusions of all proposed CUDA algorithms for some parameters.

Mathematics Subject Classification: 30C40, 65Y05, 68M20, 68W10

Keywords: Kernel density estimation, parallel computing, parallel programming, GPU computing, CUDA

1 Introduction

Nonparametric methods which do not require distributional assumptions, are one of the most important for applied data analysis, modeling and inference. One of its popular tools is kernel density estimation (KDE) and it has been successfully used to a large number of applications including computational econometrics, market analysis and computational linguistics to name but a few. There are numerous publications and references about the kernel density estimation methods mostly concerning theoretical and practical aspects of the estimation methods; see for example Silverman [22], Wand and Jones [24] and Klemel [10].

Kernel density estimation methods are typical of computational order $O(n^2k)$ where n is the number of observations and k is the number of variables and in many cases the data sets are becoming larger in recent years and these kernel estimation methods are becoming more computer intensive as econometricians estimate more complicated models and utilize more sophisticated estimation techniques. Methods of data-based bandwidth selection such as cross-validation have also high computational requirements [5].

Few approximation techniques have been proposed for reducing the huge computational requirements of kernel density estimation methods. The first of them, proposed by Silverman [23], uses Fast Fourier Transform (FFT). The other one applies Fast Gauss Transform (FGT) as suggested by Elgamall [6].

An alternative way to satisfy the computational demands of kernel estimation methods is to use the parallel computing. The most important idea of parallel computing is to divide a large-scale problem into a number of smaller problems that can be solved concurrently on independent computers. Research in parallel computing has focused on hardware and software. The most important hardware technologies or high performance computer systems are a cluster of workstations, multicore platforms and Graphics Processing Units (GPUs). On the other hand, several well-designed software technologies are available for utilizing parallel computing hardware. For example, the development and programming of applications for cluster of workstations is based on message passing programming model using the MPI library. Moreover, the parallelisation of applications on multicore architectures is based on thread programming model using the Pthreads, OpenMP, Intel Cilk++ and Intel TBB frameworks. Finally, the implementation of an application on GPUs is based on data parallel programming model using CUDA and OpenCL frameworks.

There are many references about parallel computing for related non-parametric and econometric methods and applications; see for example, Adams et al [1] and Greel and Coffe [4] for a review and the monographs by Kontoghiorghes [11, 12] treats parallel algorithms for statistics and linear econometric models. However, in the field of the parallelisation of kernel density methods there are a few research works. For example, Racine [20] presented a parallel implementation of kernel density estimation on a cluster of workstations using MPI library. Further, Creel [3] implemented the kernel regression method in parallel on a cluster of workstations using MPI toolbox (MPITB) for GNU Octave [7]. Moreover, Lukasik [13] presented three parallelisations for kernel estimation, bandwidth selection and adaptation on a cluster of computers using MPI programming model. Recently, Michailidis et al [14] presented an extended and unified study of implementations for kernel density estimation on multicore platform using different programming models such as Pthreads, OpenMP, Intel Cilk++, Intel TBB, SWARM and FastFlow. The parallelisation of kernel estimation methods of previous papers is based on the data partitioning technique where each computer or processor core executes the same operations on different portions of a large data set.

Based on research background, there isn't an extensive research work in the field of the parallelisation of kernel estimation methods on GPUs accelerators. For this reason, in this paper we present a naive and two optimised CUDA algorithms for the two kernel estimation methods such as univariate and multivariate case. These two optimised CUDA algorithms are based on the use of shared memory tiles and loop unrolling techniques. Moreover, the GPU platform is characterized as a flexible because a variable number of concurrent threads can be executed and there is a dynamic assignment of local resources, such as registers and local memory to threads [21]. Based on this characteristic of flexibility allow us to present an extended and exploratory experimental study of the proposed CUDA algorithms in order to find the optimal performance with several values of parameters such as number of threads per block, tile size, loop unroll level, number of variables and data size. Finally, in this paper we present a comparative study among the three CUDA algorithms (i.e. the naive and two optimised versions) for the parallelisation of the two kernel estimation methods.

An outline of the rest of the paper is as follows. In Section 2, a GPU architecture as well as the CUDA programming model is presented and in Section 3 there are sequential and CUDA algorithms for two kernel density estimation methods. In Section 4 the experimental methodology including the GPU platform, experimental parameters and performance metrics for the evaluation of the algorithms is presented. In Section 5, an extended performance evaluation of the three CUDA implementations is described. Finally, in Section 6 the final conclusions are presented.

2 GPU Architecture and GPU Programming

A heterogeneous computer system consists of two different types of components: a host that is a CPU multicore and the devices that are GPU massively parallel manycore cards. The GPU is connected to the CPU multicore system via 16-lane PCI Express (PCIe) 2.0 bus to provide 16 GB/s transfer rate, peak of 8 GB/s in each direction. The data transfer rate between the GPU chip and the GPU memory is more higher than the PCIe bandwidth.

In the following subsections, we introduce the hardware architecture model and the software programming model of the GPU.

2.1 GPU Architecture

More details for the historical evolution of GPU architecture from the graphics pipeline into a full scalable parallel programmable processors, the readers are referred to [19, 18]. Nvidia developed an unified GPU architecture that supports both graphics and general purpose computing. In general purpose computing, the GPU is viewed as a massively parallel processor array contains many processor cores. Figure 1 presents a typical GPU architecture from Nvidia [16]. The GPU consists of an array of streaming multiprocessors (SMs). Each SM consists of a number of streaming processors (SPs) cores. Each SP core is a highly multithreaded, managing a number of concurrent threads and their state in hardware. Further, each SM has a SIMD (Single-Instruction Multiple-Data) architecture: at any given clock cycle, each streaming processor core performs the same instruction on different data. Each SM consist of four different types of on-chip memory, such as registers, shared memory, constant cache and texture cache. Constant and texture caches are both read-only memories shared by all SPs. On the other hand, the GPU has a off-chip memory, namely global memory (to which Nvidia refers as the device memory) in which has long access latency. More details for these different types of memory can be found in [16, 21]. For our numerical experimets, we take Nvidia GeForce GTX 280 in which has 30 SMs, and each SM has 8 SPs, resulting in a total of 240 processor cores. Moreover, each SM has 16 KB shared memory and 16 KB registers.

2.2 GPU Programming Model

Programming GPUs is qualitatively different than programming multicore CPUs. For this reason, parallel to the hardware, developed many software environments for GPU computing in order to develop general purpose computing applications such as BrookGPU, Microsoft's Accelerator, RapidMind, PeakStream, AMD CTM/HAL, Nvidia CUDA, OpenCL and DirectCompute.

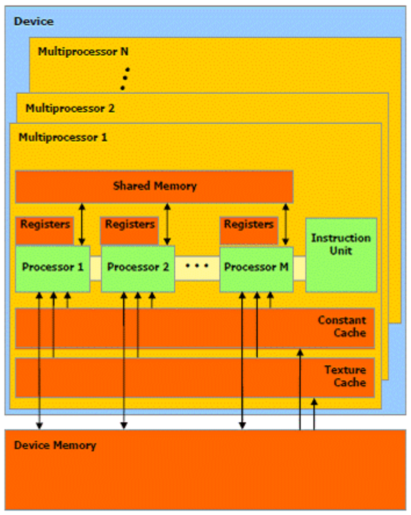


Figure 1: Typical GPU architecture from Nvidia [16]

More details for the history of software environments for GPU computing have been summarized in [18, 2]. We focus on Nvidia Compute Unified Device Architecture (in short, CUDA) programming model and software environment which allows programming GPUs for general purpose applications via minimal extension of the C programming language. The basic programming goal of CUDA is to support heterogeneous computing model that the sequential parts of an application are executed on the CPU and the computationally-intensive parts are executed on the GPU.

To map large-scale problems on a GPU parallel architecture using the CUDA programming model, the programmers follow the two-level data parallel decomposition scheme [15]: the problem is partitioned into coarse sub-problems or blocks that can be computed independently in parallel and afterwards each sub-problem or block further is partitioned into finer sub-tasks that can be computed cooperatively in parallel. The above two level parallel scheme is compatible to the GPU architecture: streaming multiprocessors compute coarse sub-problems and parallel threads compute finer sub-tasks.

The developer writes a unified program C for CUDA in which consists of host code running on CPU and kernel code running on a device or GPU. The host code is a simple code C that implements some parts of an application that exhibit little or no data parallelism. The kernel code is a code using CUDA keywords that implements some parts of an application exhibit high amount of data parallelism. The kernel code implements the computation for a single thread and is usually executed on GPU by a set of parallel threads. All parallel threads execute the same code of the kernel with different data. Threads are organized in a hierarchy of grids of thread blocks. A thread block is a set of concurrent threads that can cooperate via barrier synchronization and access to a shared memory which is only visible to the thread block. Each thread in a thread block has a unique thread ID number `threadIdx`. Thread ID can 1, 2, or 3 dimensional. Usually there can be 512 threads per block. Thread blocks are grouped into a grid and are executed independently. Each block in a grid has a unique block ID number `blockIdx`. Block ID can be 1 or 2 dimensional. Usually there are 65535 blocks in a GPU. The programmer invokes the kernel, specifying the number of threads per block and the number of blocks per grid. We must note that prior the invoking the kernel, all the data required for the computations on GPU must be transferred from the CPU memory to GPU (global) memory using CUDA library functions. Invoking a kernel will hand over the control to the GPU and the specified kernel code will be executed on this data.

Another important concept in CUDA is that the thread blocks are serially assigned for execution on each SM. These blocks are then further divided into groups called warps, each containing 32 parallel threads and is the scheduling unit of each SM. When a warp stalls, the SM can swap for another warp to execute. A warp performs one instruction at a time, so full performance can only be achieved when all 32 parallel threads in the warp have the same execution path.

Closely related to the thread hierarchy is the memory hierarchy. Each thread uses registers, each thread block uses shared memory and each application or grid uses global memory. The registers and shared memory are the fastest memories and are located on the on-chip random-access memory, but their sizes are limited. Registers are allocated by compiler, whereas shared memory is allocated by the developer. The global memory is the slowest memory and is located on the off-chip dynamic random-access memory.

Naive implementations of computationally intensive applications on GPU platform using CUDA model give important speed-ups over existing CPU, but these are often suboptimal and do not uses the hardware resources to a satisfactory degree. Achieving scalable and high performance applications on the GPU architecture using CUDA, the programmers should follow some optimization programming strategies and guidelines such as latency hiding, careful use of three layers of memory hierarchy and other similar strategies [2, 9, 21].

In addition, Nvidia offers functional accelerated libraries, like cuBLAS, cuFFT and Thrust which provide simple standard interface to often used routines for linear algebra, Fast Fourier Transform, data parallel primitives, respectively. Finally, there are useful tools that help the software development such as the compiler nvcc, the debugger cudagdb, the profiler cudaprof and documentation [16].

3 Kernel Density Estimation: Sequential and CUDA Algorithms

In this section we give the description of sequential kernel density estimation methods both the univariate and multivariate cases and we also discuss how they can be parallelised using the CUDA programming model.

3.1 Sequential Algorithms

We begin with the simplest kernel estimator that is called a univariate density estimator. Consider a random vector $x = [x_1\, x_2 \ldots x_n]^T$ of random variable x of length n. Drawing a random sample of size n in this setting means that we have n observations of the random variable x and x_i is denoted i observation of the random variable x. The goal is to estimate the kernel density of the random variable $x = [x_1\, x_2 \ldots x_n]^T$ that originally proposed by Rosenblatt which is defined as [8, 20]

$$\hat{f}(x_i) = \frac{1}{n}\sum_{j=1}^{n}\frac{1}{h}K(\frac{x_i - x_j}{h}), i = 1, 2, \ldots n \tag{1}$$

where $\hat{f}$ is a probability density function, $K(z)$ is the kernel function that satisfies $\int K(z)\mathrm{d}z = 1$ and some other regularity conditions depending on its order, and h is a bandwidth or the window width satisfying $h \to 0$ as $n \to \infty$. We must note that the bandwidth which is used in formula 1 is fixed.

A sequential algorithm in C language for a univariate fixed bandwidth density estimator shown in Algorithm 1.

Algorithm 1: Sequential univariate kernel estimation algorithm

```
1 for $i \leftarrow 0$ to $n$ do
2     $sum\_ker \leftarrow 0.0$
3     for $j \leftarrow 0$ to $n$ do
4         $sum\_ker \leftarrow sum\_ker + kernel((x[i] - y[j])/h)/h$
5     $pdf[i] \leftarrow sum\_ker/(double)n$;
```

We must note that the `kernel` is a C user-defined function which contains the formula of the Gaussian kernel function. Moreover, the vectors *pdf*, x and y are implemented as one-dimensional arrays.

Extension of the above approach to multivariate density estimation is straightforward so that involving k random variables. Consider a k-dimensional random vector $x = [x_1, \ldots, x_k]^T$ where $x_1, \ldots, x_k$ are one-dimensional random variables. Drawing a random sample of size n in this setting means that we have n observations for each of the k random variables, $x_1, \ldots, x_k$. Suppose that we collect the ith observation of each of the k random variables in the vector x_i, i.e. $x_i = [x_{i1}, \ldots x_{ik}]^T$ for $i = 1, 2, \ldots n$, where x_{ij} is the ith observation of the random variable x_j. We adapt the univariate kernel density estimator to the k-dimensional case using a product kernel. Therefore, the multivariate kernel density estimator is defined as [8]

$$\hat{f}(x_i) = \frac{1}{n} \sum_{j=1}^{n} \{ \prod_{d=1}^{k} \frac{1}{h_d} K(\frac{x_{id} - x_{jd}}{h_d}) \}, i = 1, 2, \ldots n \tag{2}$$

In the multivariate case, we consider that the bandwidth is equal in all dimensions, i.e. $h = h_1 = h_2 \ldots h_d$, where $d = 1, 2, \ldots, k$. A sequential algorithm in C language for a multivariate fixed bandwidth density estimator shown in Algorithm 2.

Algorithm 2: Sequential multivariate kernel estimation algorithm

```
1 for $i \leftarrow 0$ to $n$ do
2     $sum\_ker \leftarrow 0.0$
3     for $j \leftarrow 0$ to $n$ do
4         $prod\_ker \leftarrow 1.0$
5         for $k \leftarrow 0$ to $nvar$ do
6             $prod\_ker \leftarrow prod\_ker * kernel((x[i][k] - y[j][k])/h)/h$
7         $sum\_ker \leftarrow sum\_ker + prod\_ker$
8     $pdf[i] \leftarrow sum\_ker/(double)n;$
```

The vector *pdf* is implemented as one-dimensional array whereas the matrices x and y are implemented as two-dimensional arrays. The variable *nvar* is the number of random variables. We must note that in this paper we implement the univariate and multivariate fixed bandwidth density estimator with Gaussian kernel function. These two kernel estimation methods share a common feature - they involve $O(n^2k)$ computations in contrast to, say, the univariate density estimation that involve only $O(n^2)$ computations (in this case, $k = 1$).

3.2 Naive CUDA algorithms

Starting from the CUDA implementation of the univariate kernel estimation method, we observe that sequential algorithm 1 consists of an outer for loop where each iteration calculates the probability density function of i element of vector *pdf* (i.e. each iteration calculates the kernel summation and the final sum is stored in *pdf*[i]) and is independent of all the others. For implementing of the univariate kernel estimation method in CUDA, we follow the two-level data parallel scheme: the result vector *pdf* is partitioned into coarse blocks of equal size $\lceil n/t \rceil$ (where n is the number of elements of vector *pdf* and t is the number of blocks) and each block further is partitioned into finer sub-tasks where each sub-task performs the calculation of the individual kernel summation or the probability density function of i element of the vector *pdf*. Based on the above, we assign a coarse block to each thread block in a one-dimensional grid and we also assign a sub-task to each individual thread within a one-dimensional thread block. For this reason, an iteration of the outer loop i of Algorithm 1 is replaced by an independent thread with identifier *id* where *id* is the global identity of thread on a grid of threads. This global identity is calculated based on its thread and block IDs so that each thread based on the *id* executes the desired calculation on the corresponding element of vector *pdf*. Therefore, the serial algorithm 1 is transformed into the naive CUDA kernel for a thread, as shown in Algorithm 3.

Algorithm 3: CUDA univariate kernel estimation algorithm

```
1 $id \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
2 $sum\_ker \leftarrow 0.0$
3 for $j \leftarrow 0$ to $n$ do
4     $sum\_ker \leftarrow sum\_ker + kernel((x[id] - y[j])/h)/h$
5 $pdf[id] \leftarrow sum\_ker/(double)n;$
```

The variables *id* and *sum_ker* are allocated by compiler as registers and the vectors x, y and *pdf* are allocated as device or global memory.

Similarly, we follow the above parallelisation strategy for implementing of the multivariate kernel estimation method, as shown in Algorithm 4. In this case, an iteration of the outer loop i of Algorithm 2 is replaced by an independent thread with identifier *id*. Each thread calculates a product and a sum kernel and the final sum is stored in the corresponding element *id* of the vector *pdf*. We must note that the two-dimensional arrays x and y of Algorithm 2 are transformed into two one-dimensional arrays so that these arrays can handled by the CUDA model. Finally, the variables *id*, *sum_ker* and *prod_ker* are allocated by compiler as registers whereas the vectors x, y and *pdf* are allocated in the device or global memory.

There is data transfer between the CPU memory (host) to GPU (global or device) memory both before and after kernel invocation. So, the CPU allocates space over the global memory of GPU and transfers data required by kernels. At the end of computations, results are transferred to the CPU memory. Further, we specified one-dimensional number of blocks per grid and one-dimensional number of threads per block in the kernel invocation.

Algorithm 4: CUDA multivariate kernel estimation algorithm

```
1 $id \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
2 $sum\_ker \leftarrow 0.0$
3 for $j \leftarrow 0$ to $n$ do
4     $prod\_ker \leftarrow 1.0$
5     for $k \leftarrow 0$ to $nvar$ do
6         $prod\_ker \leftarrow prod\_ker * kernel((x[id*nvar+k]-y[j*nvar+k]/h)/h$
7     $sum\_ker \leftarrow sum\_ker + prod\_ker$
8 $pdf[id] \leftarrow sum\_ker/(double)n;$
```

3.3 CUDA optimisations

In next two subsections, we discuss two CUDA optimisation techniques, namely register and shared memory tiles in order to reduce the global memory accesses of the two naive CUDA implementations and the loop unrolling to improve further the performance.

3.3.1 Register and shared memory tiles optimisation

The global data accesses of the two naive CUDA algorithms are coalesced memory accesses when global memory addresses accessed simultaneously by multiple threads in the same warp in a contiguous way and can be aggregated into a single data access. In this way, there is a reduction in the global memory access. Nonetheless, we observe that there are two significant disadvantages from naive CUDA algorithm 3. First, each thread with id reads the same id element of the vector x, n times from the slow global memory and second all the threads within the same block read frequently the same whole vector y, n times from the global memory, i.e. the vector y is shared by all the threads within the same block. These two disadvantages slow down the performance of CUDA algorithm 3 significantly. For this reason, we can eliminate these two disadvantages exploiting the memory hierarchy of the CUDA model in order to reduce further the global memory accesses. Therefore, the first disadvantage can be eliminated by the effective use of the registers. More specifically, we declare a variable xr to each thread in order to load once the corresponding id element of the vector x. This variable will be allocated automatic by the compiler as a register.

The second disadvantage can be eliminated by the shared memory use so that the vector y accessed at the tile level reside on the shared memory. The whole vector y is partitioned into blocks of size $BSIZE$ elements. A block with $BSIZE$ contiguous elements is called tile and the $BSIZE$ is the tile size. Each block is loaded on the shared memory sequentially tile by tile. The loading of each tile on the shared memory is as follows: all the threads within the same thread block load their corresponding elements of the vector y cooperatively. This tile-loading procedure is performed when the tile size is equal the number of threads (i.e. $THREADS$) in a thread block. When the tile size is greater than the number of threads of a thread block, the above tile-loading procedure can be performed in iterative and successive phases. The implementation of the tile-loading procedure in CUDA requires the allocation of memory for the tile in the shared memory and the execution of data transfer from global to shared memory. The following piece of code C contains statements for allocating a array in the shared memory using the `__shared__` keyword and statements for transferring of data from the global memory (y vector) to the shared memory (yr vector):

```
__shared__ float yr[BSIZE];

for(j = 0; j < n; j += BSIZE) {
    for(i = 0; i < BSIZE; i += THREADS)
        yr[threadIdx.x + i] = y[(i + (threadIdx.x + j))];
    ...
}
```

Once each tile loaded into the shared memory, all operations within the tile are calculated in the shared memory and registers. Based on the above observations, naive CUDA algorithm 3 is transformed into a shared memory optimisation kernel for a thread, as shown in Algorithm 5.

Algorithm 5: CUDA optimisation algorithm for univariate kernel estimation

```
1  $\_\_shared\_\_\ float\ yr[BSIZE]$
2  $id \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
3  $sum\_ker \leftarrow 0.0$
4  $xr \leftarrow x[id]$
5  for $j \leftarrow 0$ to $n$ with step $BSIZE$ do
6      for $i \leftarrow 0$ to $BSIZE$ with step $THREADS$ do
7          $yr[threadIdx.x + i] \leftarrow y[(i + (threadIdx.x + j))]$
8      for $k \leftarrow 0$ to $BSIZE$ do
9          $sum\_ker \leftarrow sum\_ker + kernel((xr - yr[k])/h)/h$
10 $pdf[id] \leftarrow sum\_ker/(double)n;$
```

As far as the mulivariate case is concerned, naive CUDA algorithm 4 shares the same disadvantages with CUDA algorithm 3. In the other words, each thread loads the same row id of the array x, $n \cdot k$ times from the global memory. Moreover, all the threads within the same block read the same whole matrix y, $n \cdot k$ times again from the global memory. For solving these global memory access problems, we adapt the two above proposed solutions to the multivariate kernel estimation method. For the first problem, we declare a array xr of size k elements to each thread with id as a register in order to load once the corresponding row id of the matrix x. This register declaration requires the use of `register` keyword.

For the second problem, we use rectangular tiles in the shared memory to minimize the global memory accesses. The whole matrix is partitioned into tiles of size $BSIZE \times k$ elements (where k is the number of variables). Each block is loaded on the shared memory vertically tile by tile. The loading of each tile on the shared memory is similar to this the univariate kernel method, but we load tile of size $BSIZE \times k$ elements. After loading of each tile on the shared memory follows the tile calculations within the shared memory. Therefore, CUDA algorithm 4 is transformed into a shared memory optimisation kernel for a thread, as shown in Algorithm 6.

Algorithm 6: CUDA optimisation algorithm for multivariate kernel estimation

```
$register\ float\ xr[NUM\_VARS]$
$\_\_shared\_\_\ float\ yd[BSIZE][NUM\_VARS]$
$id \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
for $k \leftarrow 0$ to $nvar$ do
    $indx \leftarrow k + id * nvar$
    $xr[k] \leftarrow x[indx]$
$sum\_ker \leftarrow 0.0$
for $j \leftarrow 0$ to $n$ with step $BSIZE$ do
    for $i \leftarrow 0$ to $BSIZE$ with step $THREADS$ do
        for $k \leftarrow 0$ to $nvar$ do
            $yd[threadIdx.x + i][k] \leftarrow$
            $y[(i * nvar) + (k + (threadIdx.x + j) * nvar)]$
    for $jj \leftarrow 0$ to $BSIZE$ do
        $prod\_ker \leftarrow 1.0$
        for $k \leftarrow 0$ to $nvar$ do
            $prod\_ker \leftarrow prod\_ker * kernel((xr[k] - yd[jj][k]/h)/h$
        $sum\_ker \leftarrow sum\_ker + prod\_ker$
$pdf[id] \leftarrow sum\_ker/(double)n;$
```

The performance of algorithms 5 and 6 give the best performance because they reduce further the global memory access. However, the amount of registers and shared memory used by each thread block limits the number of thread blocks and the number of concurrent threads executed on each SM. For this reason, we must select appropriate tile sizes in order to maximize the performance.

3.3.2 Loop unrolling optimisation

From CUDA algorithm 6, two innermost loops such as lines 10-11 and lines 14-15 have added in order to perform the data tile transfer from the global memory to the shared memory and the tile calculation, respectively. These loops have a small body and constant iteration count. However, the iteration count depends on the number of variables k each time. The performance of algorithm 6 can further improved by unrolling these small inner loops by several different levels according to the number of variables. Therefore, algorithm 6 can be transformed into a loop unrolling optimisation kernel, unrolling inner loops with level 2 (i.e. for two variables), as shown in Algorithm 7.

Algorithm 7: CUDA optimisation algorithm for multivariate kernel estimation

```
1  $register\ float\ xr[NUM\_VARS]$
2  $\_\_shared\_\_\ float\ yd[BSIZE][NUM\_VARS]$
3  $id \leftarrow blockDim.x * blockIdx.x + threadIdx.x$
4  for $k \leftarrow 0$ to $nvar$ do
5      $indx \leftarrow k + id * nvar$
6      $xr[k] \leftarrow x[indx]$
7  $sum\_ker \leftarrow 0.0$
8  for $j \leftarrow 0$ to $n$ with step $BSIZE$ do
9      for $i \leftarrow 0$ to $BSIZE$ with step $THREADS$ do
10         $yd[threadIdx.x{+}i][0] \leftarrow y[(i{*}nvar){+}(0{+}(threadIdx.x{+}j){*}nvar)]$
11         $yd[threadIdx.x{+}i][1] \leftarrow y[(i{*}nvar){+}(1{+}(threadIdx.x{+}j){*}nvar)]$
12     for $jj \leftarrow 0$ to $BSIZE$ do
13         $prod\_ker \leftarrow 1.0$
14         $prod\_ker \leftarrow prod\_ker * kernel((xr[0] - yd[jj][0]/h)/h$
15         $prod\_ker \leftarrow prod\_ker * kernel((xr[1] - yd[jj][1]/h)/h$
16         $sum\_ker \leftarrow sum\_ker + prod\_ker$
17 $pdf[id] \leftarrow sum\_ker/(double)n;$
```

This algorithm can easily modified unrolling inner loops for any level (i.e. any number of variables). However, higher levels of loop unrolling give the minimal loop overhead and fewer instructions but this may increase the registers use in each thread.

4 Experimental Methodology

In this section we present the experimental methodology which used in our experiments in order to compare the relative performance of the proposed CUDA algorithms presented in previous section. The parameters which is described the performance of the algorithms are: number of threads per thread block, tile size, loop unroll level, number of variables k (i.e. univariate kernel methods for $k = 1$ and multivariate kernel methods for $k > 1$) and data size n. It is known that none of the algorithms are optimal or best in all five cases. Therefore, the main goals in our experimental study are as follows:

1. to find the optimal performance of the naive CUDA algorithms against the number of threads per block under the different number of variables,
2. to find the optimal performance of the shared memory CUDA algorithms against the number of threads per block and the tile size under the different number of variables,
3. to find the optimal performance of the loop unrolling CUDA algorithms against the number of threads per block, the tile size and the loop unroll level under the different number of variables and
4. to compare the practical performance among the proposed CUDA algorithms based on the optimal values of the parameters (i.e. number of threads per block, tile size and loop unroll level) under the different number of variables and different data (sample) sizes.

The values of five parameters that we tested during our experiments are as follows: we tested the values 64, 128, 256 and 512 for the parameter of the thread block size, we also tested the values 2, 6, 14, 30 and 62 for the number of variables, we tested the tile sizes ranging from 128 to 2048 elements for the univariate kernel method and tile sizes ranging from $32 \times k$ to $1024 \times k$ (where k is the number of variables) for the multivariate kernel method, we tested the values 2, 6, 14, 30 and 62 for the parameter of the loop unroll level and finally we tested data sizes ranging from 1024 to 10,240.

The experiments were run on an Intel Corel2 Duo (E8400) with a 3.00GHz clock speed and 3 Gb of memory under Ubuntu Linux operating system version 10.04. The GPU used was the Nvidia GeForce GTX 280 with 1GB global memory, 30 multiprocessors and 240 cores, enabling the use of a maximum number of 1024 active threads per multiprocessor for a total of 30720 active threads. To get reliable experimental results, the running time results were measured as an average of 10 runs.

The implementations of serial and CUDA algorithms presented in the previous section were developed using the ANSI C programming language and were compiled using Nvidia's nvcc compiler with -O3 optimisation flag.

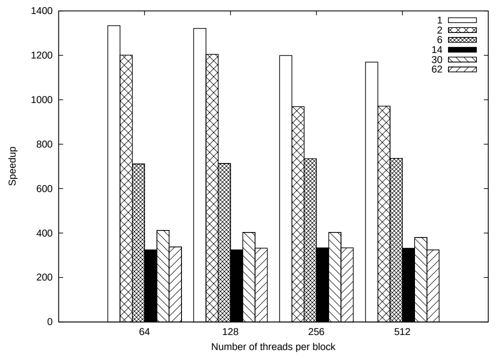
For the evaluation of the CUDA algorithms we used the GPU running time and the speed-up as measures. The GPU running time is the total time in seconds an algorithm needs to perform the calculations on the GPU platform and any transfer time between the host and the GPU and was measured using the timer functions. Finally, the speed-up is the ratio of the serial running time of an algorithm executed on the CPU platform to the GPU running time of the same algorithm executed on the GPU platform.

5 Experimental Results

In this section we present the experimental results of the proposed CUDA algorithms or kernels according to the four goals of the experimental study as we discussed in the previous section. More specifically, we present the experimental results of the CUDA algorithms according to the performance of the naive algorithms, the shared memory optimisations, the loop unrolling optimisation, and the performance comparison among the proposed CUDA algorithms based on the optimal values of the parameters (i.e. the number of threads per block, the tile size and the loop unroll level) using both univariate and multivariate kernel methods.

5.1 Performance results of naive CUDA algorithms

Figure 2 (left) presents the performance speed-ups achieved by the parallel execution of the two naive CUDA algorithms for a number of variables $k \geq 1$; i.e. for $k = 1$ the univariate kernel method and $k > 1$ the multivariate kernel method under different numbers of threads per block for a data size of 10,240. From the figure we can see that, for small numbers of variables, i.e. 1 to 2, the performance of CUDA kernels is decreased slightly as the number of threads per block is increased. More specifically, the performance is the same when the number of threads per block is 64 or 128, whereas there is a small reduction in performance speed-up when the thread block size is 256 or 512. Nonetheless, in both cases, the same number of concurrent threads is executed on each SM (i.e. achieving maximum occupancy), as shown in Figure 2 (right). This figure shows the estimated number of thread blocks that takes each SM for a given thread block size and number of variables. This estimated number of thread blocks is calculated by Nvidia's Occupancy tool [17] based on the thread block size, number of registers per thread and amount of shared memory per block. The estimated number of registers and the estimated amount of shared memory used by each kernel code are reported by the PTX code via the *pxta* option of the nvcc compiler. The number of threads executed on each SM for a given thread block size and number of variables is defined by the product of the number of blocks and the thread block size. From this figure we can



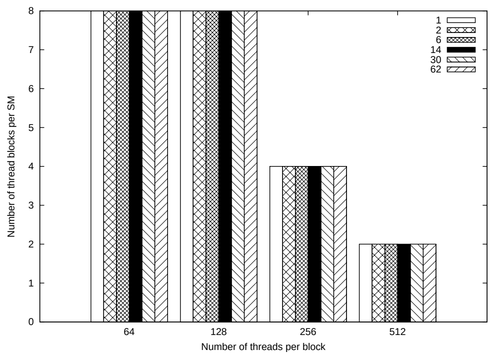


Figure 2: Performance speed-ups (left) and estimated number thread blocks (right) for different numbers of variables under different numbers of threads per block

see that each SM receives many blocks when the thread block size is small, and this seems to be better (i.e. yielding a better performance speed-up) than receiving a few blocks when the thread block size is large. On the other hand, for the remaining number of variables, i.e. 6 to 62, the performance speed-up is constant as the thread block size is increased. We also observe that there is an optimal relative performance speed-up when the thread block size reaches 128 threads for any number of variables.

Moreover, from Figure 2 (left) we can observe that, for each thread block size, there is an inverse relation between the performance speed-up of the CUDA algorithms and the number of variables. More specifically, the performance speed-up is decreased as the number of variables is increased. This is due to the fact that the workload per thread is increased as the number of variables is increased, resulting in a performance degradation even though the same number of concurrent threads is executed on each SM, as shown in Figure 2 (right).

5.2 Performance results of shared memory optimisations

Figure 3 presents the performance speed-ups achieved by the parallel execution of the CUDA shared memory optimisations for, respectively, 1, 2, 6, 14, 30 and 62 variables for a data size of 10,240. The performance results in each figure are plotted for different tile sizes in combination with the number of variables for each thread block size. Note that, as the number of variables is increased, performance speed-up bars are not shown for some values of tile size and thread block size. We will discuss this later. The observations from the first plot of Figure 3 (univariate kernel method) are as follows. For small thread block sizes such as 64 and 128 we can see that as we increase the tile size up to 256

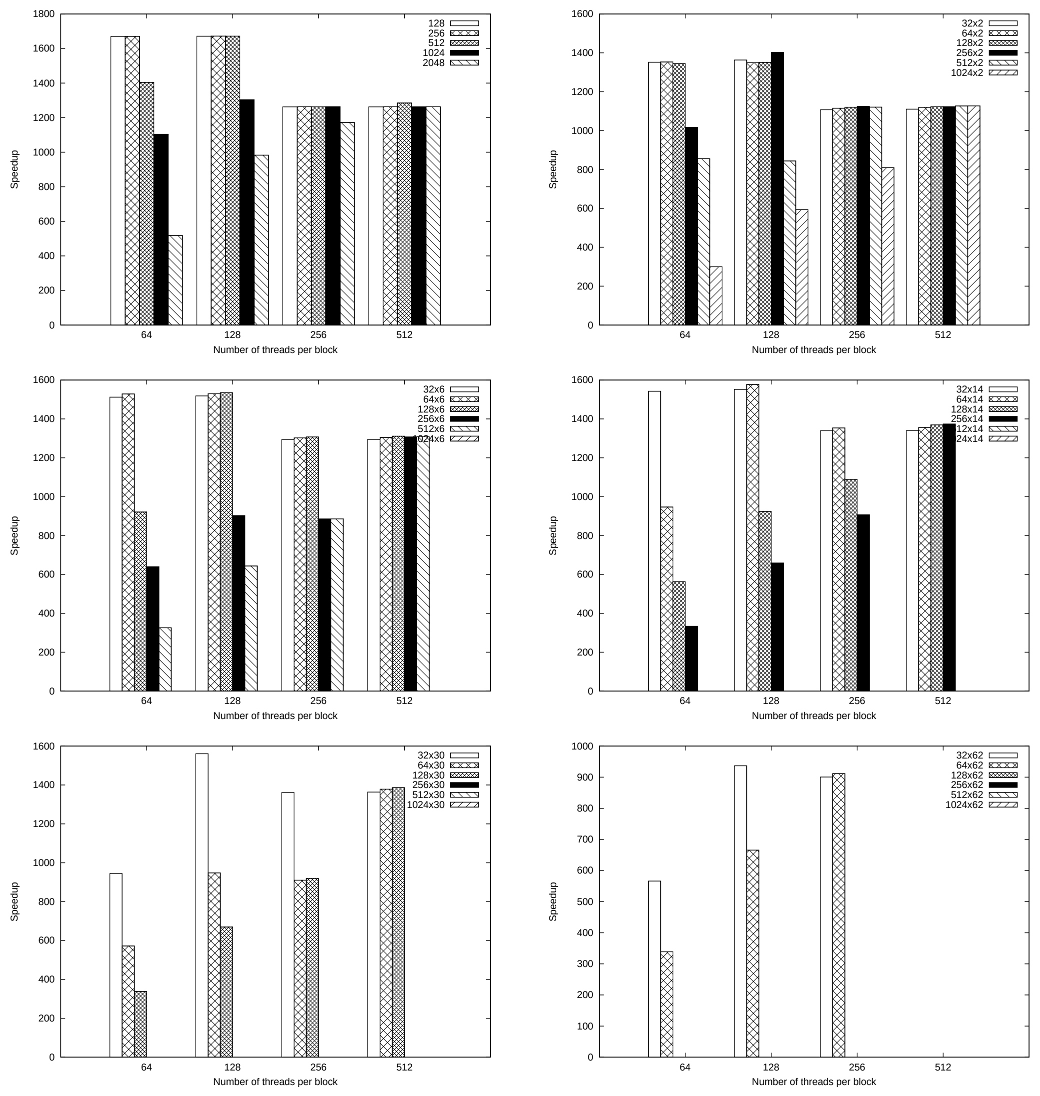


Figure 3: Performance speed-ups for different numbers of threads per block and tile sizes under the different numbers of variables 1, 2, 6, 14, 30 and 62 (from left to right)

elements, the performance improves and stabilises due to a better use of shared memory in combination with the larger number of concurrent threads executed on each SM. This fact is supported in the first plot of Figure 4, where each SM takes 8 thread blocks for thread block sizes up to 128 and tile sizes up to 256. The number of thread blocks allocated to each SM depends on the amount of shared memory used by each thread block, such that the total amount of shared memory used by all thread blocks cannot exceed 16KB. However, as we increase the tile size beyond 256 elements, the performance decreases. This is due to the fact that, as we increase tile size for small block size, the shared memory requirements are increased and the number of thread blocks allocated to each SM is reduced, as shown in the first plot of Figure 4. This has the effect of limiting the number of concurrent threads executed in each SM, and therefore limits the parallelism and the performance speed-up. For large thread block sizes such as 256 and 512 we can see that, as we increase the tile size, the performance stabilises due to a better use of shared memory in combination with the constant number of thread blocks (or concurrent threads) executed on each SM with minimal exceptions, as shown in the first plot of Figure 4. Another main reason for this constant performance speed-up is that only a few phases are required for loading large tile sizes (i.e. lines 6-7 of algorithm 5) when many threads participate in a thread block.

Furthermore, the performance of shared memory optimisation for the univariate kernel method achieves optimal results for tile sizes ranging from 128 to 512 with only a few threads per block (i.e. 64 or 128) rather than many threads per block. Small tile sizes work well with small thread block sizes because they require only a small number of successive tile-loading phases (i.e. lines 6-7 of algorithm 5), as shown in Table 1, and all threads in each block participate cooperatively in these phases to load a complete tile, resulting in a good performance speed-up. Conversely, the small tile sizes do not work well on large thread block sizes because only half of the threads participate in the tile-loading phases, whereas the remaining issue slots of a thread block (i.e. threads) remain unused, which results in performance inefficiency. On the other hand, the performance of algorithm 5 is optimal for tile sizes ranging from 1024 to 2048 elements with 256 to 512 threads per block. This is explained by the fact that the large tile sizes work well on many threads per block because these threads execute a few successive phases for loading a large tile size. However, small thread block sizes require more and more successive phases for loading large tile sizes as shown in Table 1, thus creating additional overhead time and thereby limiting the performance speed-up.

As we observed in Figure 3, as the number of variables is increased, some performance bars are not shown for some values of tile size and thread block size. This is because memory limits have been exceeded in these cases, and therefore the thread blocks are not allocated to each SM for execution. It is

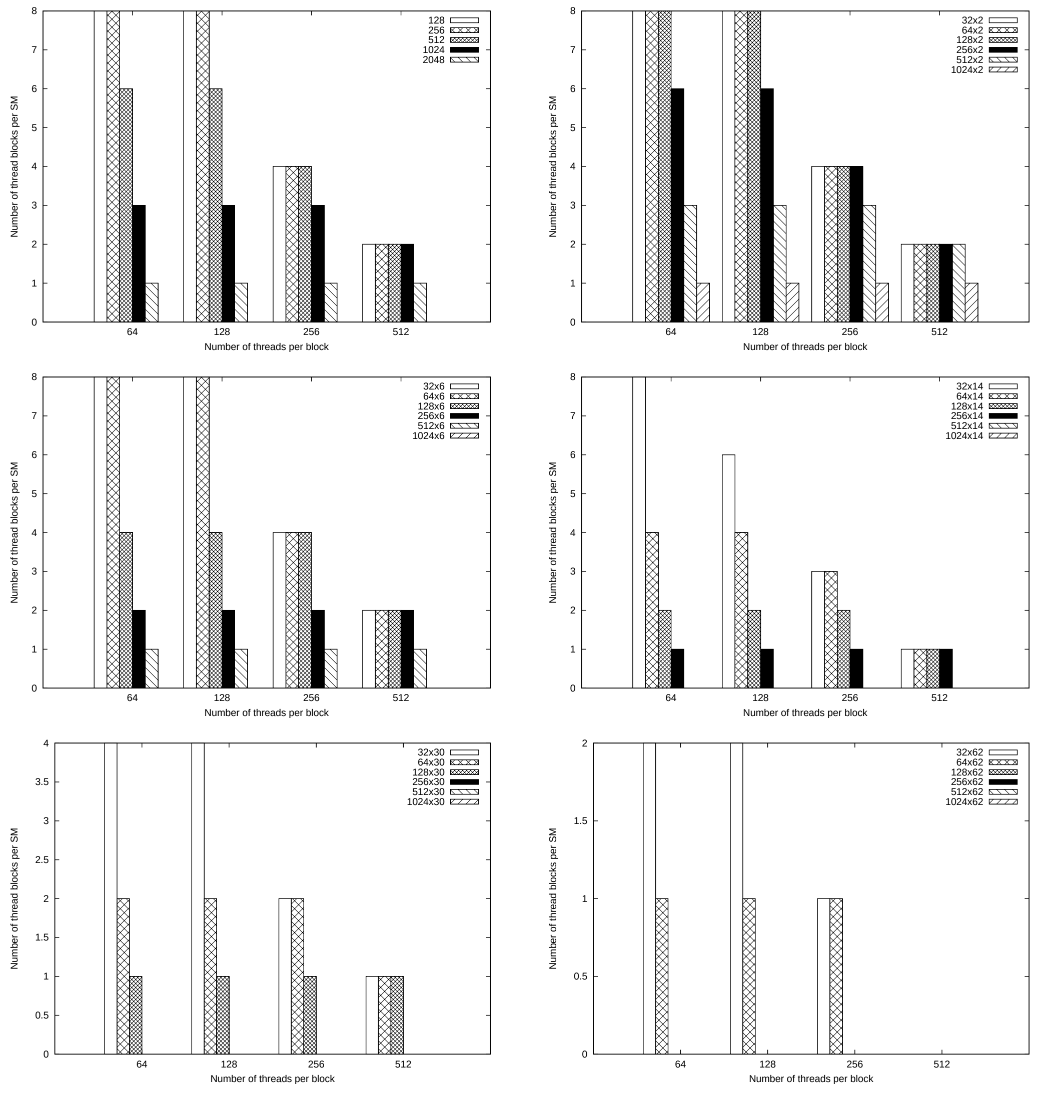


Figure 4: Estimated number of thread blocks per SM for different numbers of threads per block and tile sizes under the different numbers of variables 1, 2, 6, 14, 30 and 62 (from left to right)

Tile size \ Th. block size	64	128	256	512
128	2	1	0.5	0.25
256	4	2	1	0.5
512	8	4	2	1
1024	16	8	4	2
2048	32	16	8	4

Table 1: Number of tile-loading phases for any combination of tile size and thread block size for the univariate kernel method

known that the shape of the tile in the multivariate kernel methods consists of rows and columns. The number of rows is chosen by the user, whereas the number of columns is equal to the number of variables used by the multivariate method each time. However, we must be careful to select the appropriate tile size so as not to exceed the capacity of the shared memory. For this reason, the smaller the number of rows, the larger the number of variables, or conversely the larger the number of rows, the smaller the number of variables that will fit within the shared memory. Therefore, when selecting the appropriate tile size so as not to exceed the capacity of the shared memory, there is an inverse relation between the number of rows and the number of variables of a tile.

Similar observations apply to the performance speed-up figures for the other variables, $2, 6, 14, 30$ and 62. For small thread block sizes such as 64 and 128 we can see that as we increase the tile size up to 256×2 elements, 128×6 elements, 64×14 elements, 32×30 elements and 32×62 elements for the number of variables (2, 6, 14, 30 and 62, respectively), the performance improves and stabilises due to a better use of shared memory in combination with the larger number of concurrent threads or thread blocks executed on each SM, as shown in Figure 4. On the other hand, as we increase the tile size beyond 256×2, 128×6, 64×14, 32×30 and 32×62 elements for the number of variables (2, 6, 14, 30 and 62, respectively), the performance decreases because the number of thread blocks allocated to each SM is reduced, as shown in Figure 4. For large thread block sizes such as 256 and 512, and any number of variables, we can see that as we increase the tile size, the performance is constant with minimal exceptions, and this is due to a better use of shared memory in combination with the constant number of thread blocks executed on each SM, as shown in Figure 4. In addition, for any number of variables, we can see from Figure 3 that the small thread blocks size, i.e. 128, achieves optimal performance speed-up for small tile sizes because the small number of threads of a thread block require a few successive phases for loading these small tile sizes, as we explained earlier in the univariate case. Similarly, the best performance is achieved when large numbers of threads are used (i.e. 512) for loading large tile sizes, as explained earlier. More specifically, for thread block size 128 there is optimal performance for tile sizes of 256×2 elements for two variables, 128×6 for six variables, 64×14 for 14 variables, 32×30 for 30 variables and 32×62 for 62 variables. On the other hand, for thread block size 512 there is optimal performance for tile sizes of 1024×2 elements for two variables, 512×6 for six variables, 256×14 for 14 variables and 128×30 for 30 variables. However, from Figure 3 we can also observe that for a given problem with a specific number of variables, optimal performance is achieved when the thread block size reaches 128 with small tile sizes rather than 512 with large tile sizes.

As a first general conclusion, we can tell that, for small thread block sizes, there is an inverse relation between the performance speed-up (or the number of thread blocks) and the tile size regardless the number of variables, whereas, for large thread block sizes, there is an constant relation slightly between the performance speed-up and the tile size. Further, a second general conclusion, we can tell that optimal performance of CUDA algorithms 5 and 6 is achieved when there is a linear relation between the thread block size and tile size for any number of variables. More specifically, optimal performance of these CUDA algorithms is achieved when they process small or large tile sizes in the shared memory with the cooperation of a small or large thread block size such that the tile loading procedure in the shared memory can be accomplished in few rather than many phases.

5.3 Performance results of loop unrolling optimisation

Figure 5 presents the performance speed-ups achieved by the parallel execution of the CUDA loop unrolling optimisation for, respectively, 2, 6, 14, 30 and 62 variables, for a data size of 10,240. The experimental results aren't displayed for one variable (or the univariate method) because there is no sense in applying the technique of loop unrolling to a single variable. The performance results in each Figure are plotted for different tile sizes in combination with the number of variables for each thread block size. Note that, as the number of variables is increased, some performance bars in some values of tile size and thread block size are not shown because the memory limits are exceeded in these cases and therefore the thread blocks are not allocated to each SM for execution, as we discussed earlier.

The trend of these results is similar to those of shared memory optimisations, as discussed in the previous subsection. However, the performance degradation of the algorithms for small thread block sizes (i.e. 64 and 128) as we increase the tile size isn't due only to the increased use of shared memory per block, but also to the increased use of registers per thread because of the technique of loop unrolling. Note that we unroll the small inner loops of algorithm 7 by different levels according to the number of variables. These ag-

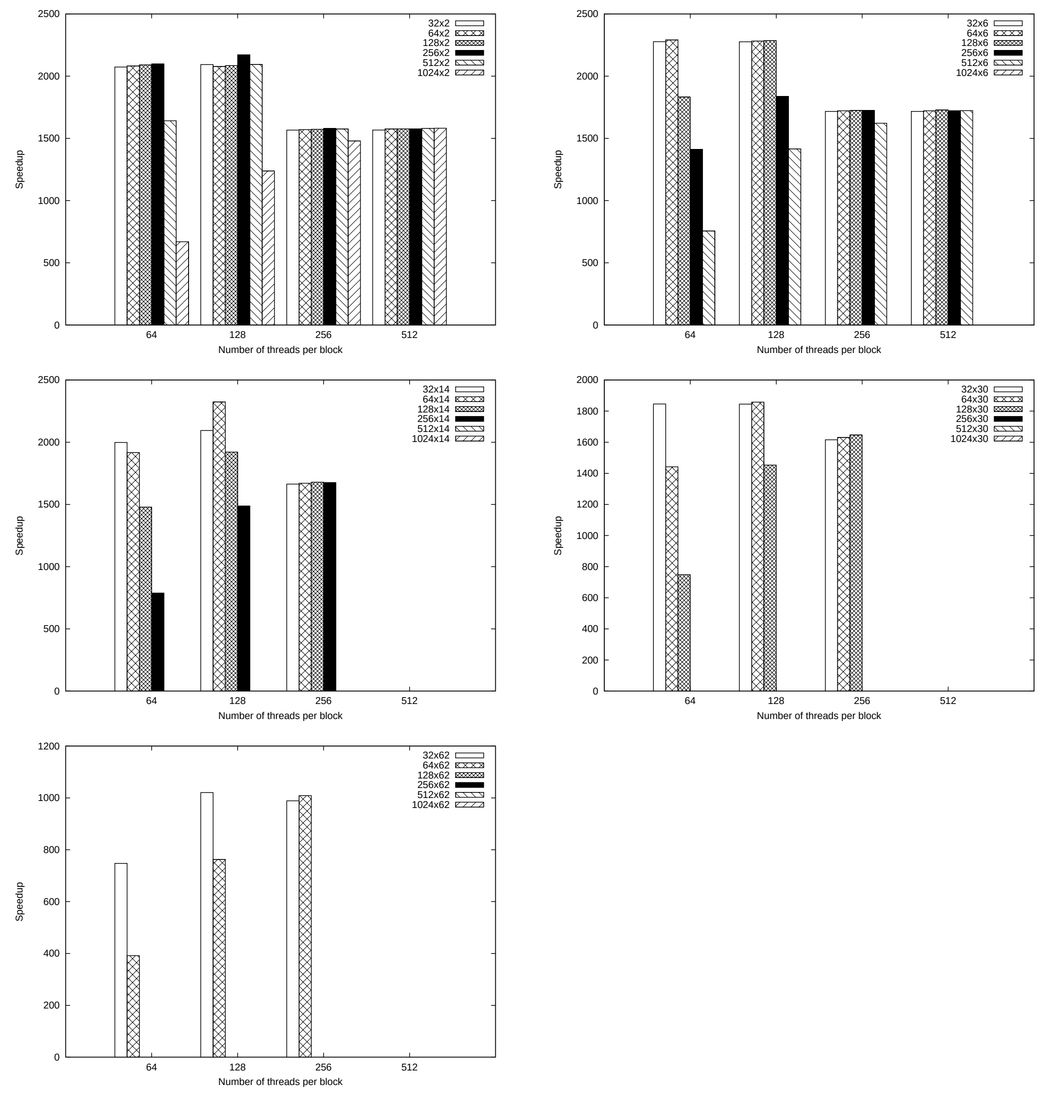


Figure 5: Performance speed-ups for different numbers of threads per block, tile sizes, loop unroll levels under the different numbers of variables 2, 6, 14, 30 and 62 (from left to right)

gressive increases in shared memory and register requirements limit the number of thread blocks and the number of threads allocated to each SM (as shown in Figure 6) quickly enough compared to the shared memory optimisations that the parallelism is small. This observation is valid for any number of variables.

Finally, we can conclude that unrolling loops by higher levels achieves higher performance speed-ups at the limited values of parameters such as the tile and the thread block sizes through the limiting of resources, i.e. registers. In other words, there is an inverse relation between the loop unroll level and the use of resources.

5.4 Performance comparison among the proposed CUDA algorithms

In this section we present performance comparisons of the proposed CUDA algorithms for univariate and multivariate kernel estimation methods. Figure 7 presents the performance speed-ups of the two CUDA algorithms, 3 and 5, over the CPU algorithm based on the optimal thread block size of 128 and the tile size of 512 elements for different data sizes. From Figure 7 we can see that the performance speed-up of both algorithms is increased as the data size is increased. Note that, for this specific combination of thread block size and tile size, algorithm 3 executes the maximum number of concurrent threads in each SM (i.e 1024), whereas algorithm 5 uses only 768 concurrent threads. Based on previous experimental results, this limited number of threads of algorithm 5 is due to the increased shared memory requirements. Although this limited number of concurrent threads of algorithm 5, the performance speed-up of algorithm 5 is greater than the speed-up of algorithm 3 as the data size is increased because there is better reuse of shared memory data for this specific combination of thread block size and tile size. More specifically, algorithm 5 achieves from 1.3X to 1.7X speed-up compared to algorithm 3.

Figure 8 presents the performance speed-ups of the three CUDA algorithms, 4, 6 and 7, over the CPU algorithm for different data sizes under different numbers of variables k. For $k = 2$ we used the optimal combination of thread block size of 128 and tile size of 256×2 elements; for $k = 6$ we used the optimal combination of thread block size of 128 and tile size of 128×6 elements; for $k = 14$ we used the optimal combination of thread block size of 128 and tile size of 64×14 elements; and for $k = 30$ and $k = 62$ we used the optimal combination of thread block size of 128 and tile size of 32×30 elements. For any number of variables, we observe from these figures that the performance speed-ups of the three algorithms are increased as the data size is increased. Moreover, the speed-ups of algorithm 6 are higher than algorithm 4 as the data size and the number of variables are increased. Although the high performance of algorithm 6, this algorithm executes the limited number of

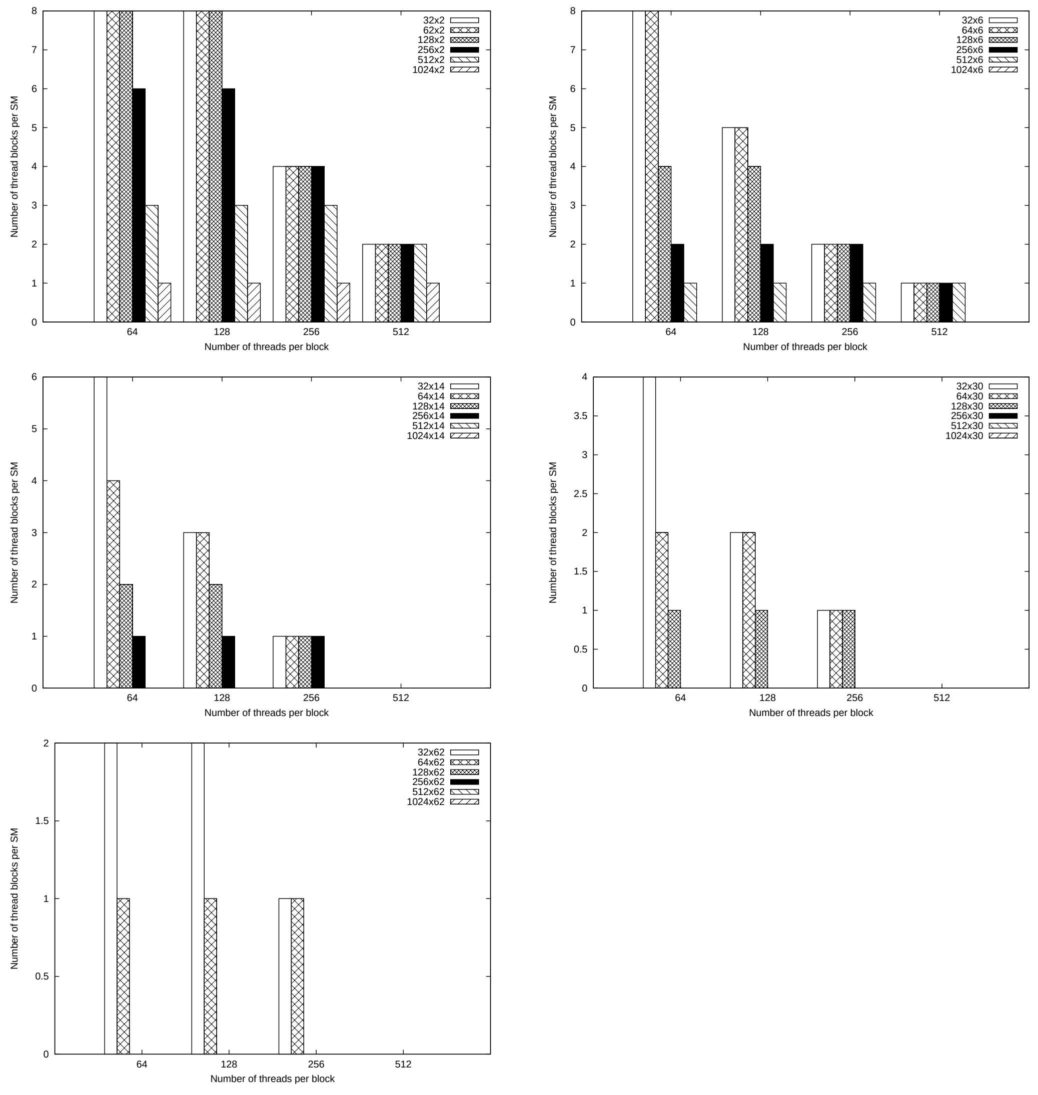


Figure 6: Estimated number of thread blocks per SM for different numbers of threads per block and tile sizes under the different numbers of variables 2, 6, 14, 30 and 62 (from left to right)

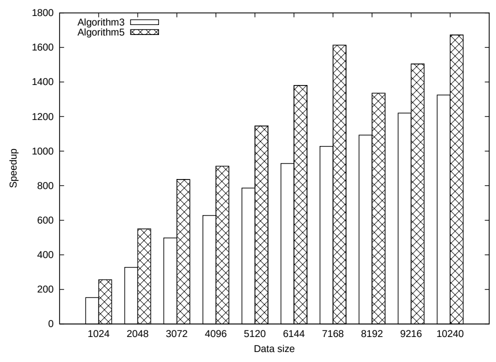


Figure 7: Performance speed-ups of the two CUDA algorithms 3 and 5 over the CPU algorithm for different data sizes for number of variables 1

concurrent threads in each SM such as 768, 512, 512, 512 and 256 threads for 2, 6, 14, 30 and 62 variables, respectively, instead of 1024 threads that uses algorithm 4. This limited number of threads is due to the increased shared memory requirements as the tile size in combination with the number of variables is increased, as we discussed earlier. Further, the performance speed-up gap between algorithms 4 and 6 widens in favour of algorithm 6 as the number of variables and the data size are increased. This is due mainly to two reasons: first, there is better reuse of shared memory data for tile sizes where the number of rows is small and the number of columns is large in combination with the small number of phases required for loading such a tile with a few rows (lines 9-11 of algorithm 6). Secondly, the increased use of fast registers by each thread for storing the k elements (the number of variables) that correspond to a row of the matrix x (i.e. lines 4-6 of algorithm 6).

Finally, for each number of variables, we can see that the performance of algorithm 7 is higher than the other two CUDA algorithms as the data size is increased, and this is due to the increased use of the fast registers compared to the other two algorithms. This increased register use is due to the loop unrolling technique, and therefore this technique appears to yield the best performance speed-ups. Furthermore, the performance of algorithm 7 achieves higher speed-ups as the number of variables is increased up to 30 variables, and this is due to the increased use of registers. This increase occurs because we unroll the inner loops by different levels according to the number of variables. Consequently, the algorithm is executed quickly. Algorithm 7 achieves higher speed-ups compared with algorithm 4 and smaller speed-ups compared with algorithm 6 as shown in Figure 9 for 2, 6, 14, 30 and 62 variables, respectively. Although the high performance of algorithm 7, this algorithm executes the limited number of concurrent threads in each SM compared with algorithms 4 and 6 and this is due to the aggressive shared memory and register requirements. Finally, we observe that the performance of the three CUDA

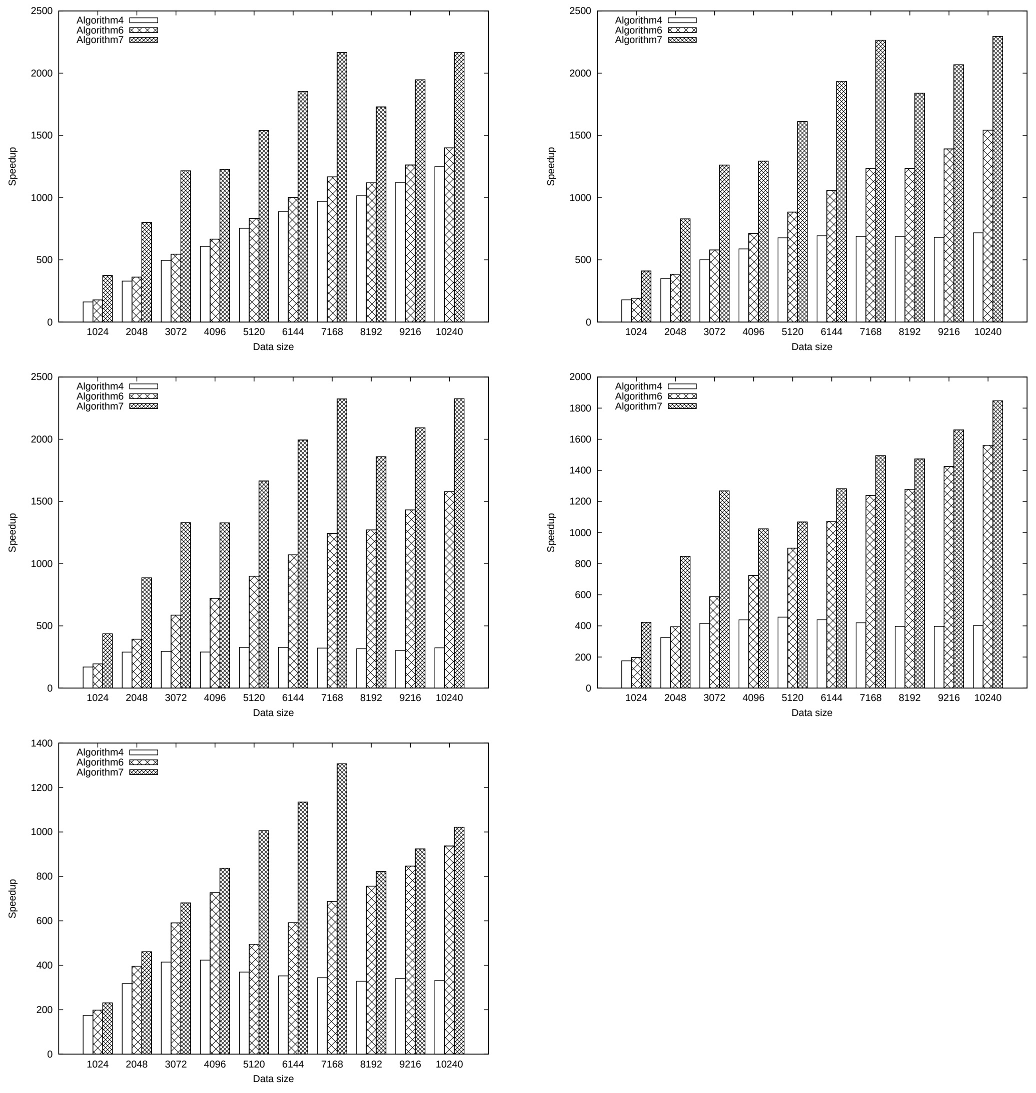


Figure 8: Performance speed-ups of the three CUDA algorithms 4, 6 and 7 over the CPU algorithm for different data sizes under different numbers of variables k 2, 6, 14, 30 and 62 (from left to right)

algorithms is limited as the number of variables is increased for a given data size.

6 Conclusions

In this paper we discussed the three CUDA algorithms for parallelisation of the univariate and multivariate kernel estimation methods. Moreover, we presented an extended and exploratory performance study of the CUDA algorithms in order to find the optimal performance according to several experimental parameters such as the number of threads per block, the tile size, the loop unroll level, the number of variables and the data size. Based on experimental results we obtain the following general conclusions. First, there is a significant relationship between the performance of all proposed CUDA algorithms and the number of variables regardless of the thread block size whereas there is a constant relationship between the performance of the naive CUDA algorithms and thread block size regardless of the number of variables. Second, there is a negative relationship between the performance (or the number of thread blocks) of the two optimised algorithms and the tile size regardless the number of variables. Third, there is a positive relationship between the thread block size and tile size regardless of the number of variables in the optimal performance of two proposed optimised CUDA algorithms (i.e. shared memory optimisation and loop unrolling). Fourth, the performance of the loop unrolling algorithm depends on the loop unroll level (or the number of variables) significantly, and this also relates to the use of resources. In other words, the higher the loop unroll level, the more resources (i.e. registers) used, resulting in limited or no parallelism. Fifth, the performance gains of all proposed CUDA algorithms are significant high over the CPU versions whereas the performance gains of the two optimised algorithms are small over the naive CUDA versions. However, these performance gains are achieved by a difficult and careful GPU programming effort for a beginner in order to develop optimisation schemes such as shared memory tiles and loop unrolling. Finally, the best performance among the proposed CUDA algorithms based on the optimal values of the same parameters is the shared memory algorithm 5 for the univariate kernel case and the loop unrolling algorithm 7 for the multivariate case with the minimal number of concurrent threads executed by each SM. This high performance of algorithm 7 is achieved based on the appropriate choice of parameters: thread block size, tile size and number of variables; otherwise it achieves limited or no parallelism.

The present experimental study could be extended in order to test other kernel functions such as epanechnikov, rectangular, triangular and cosine in the univariate and multivariate kernel estimation methods.

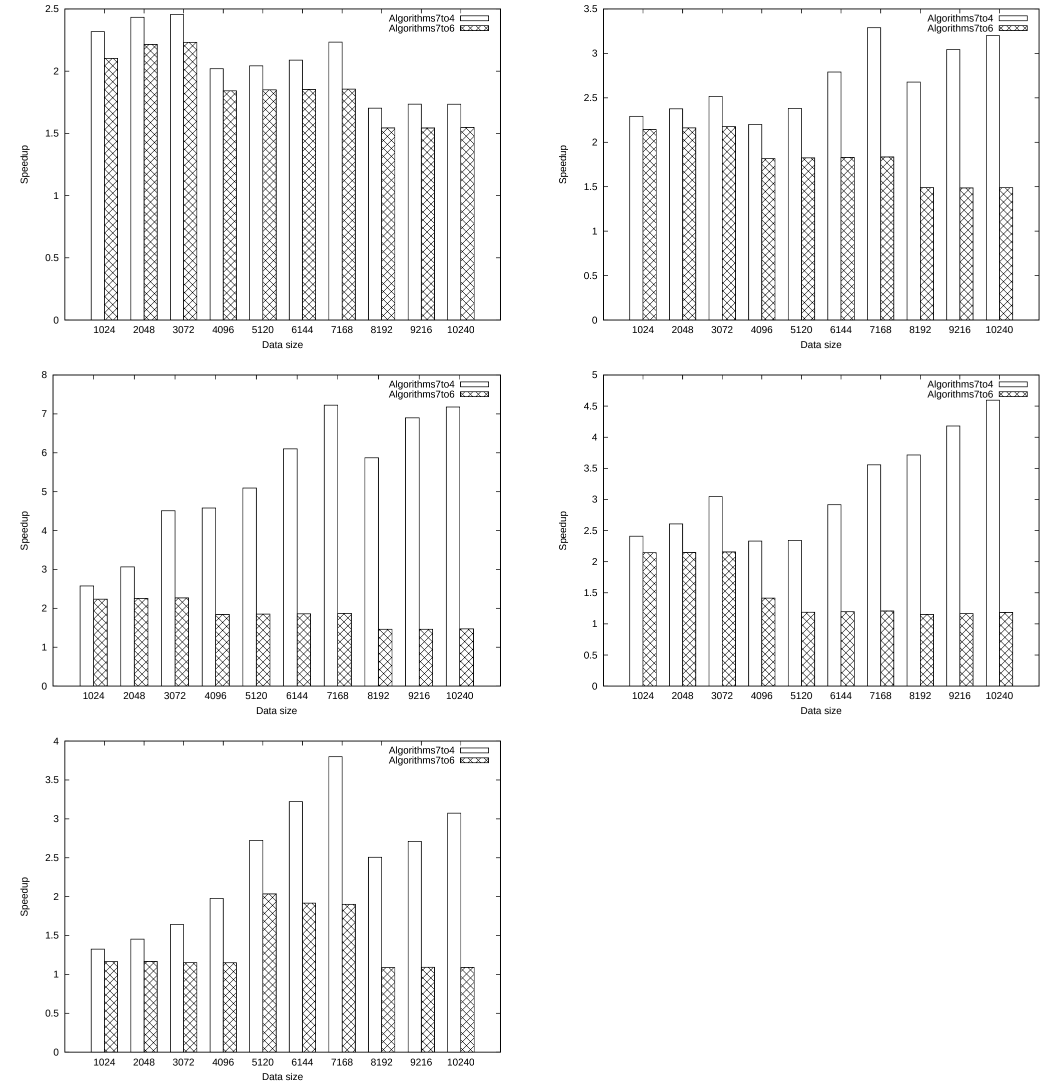


Figure 9: Performance speed-ups of the CUDA algorithm 7 over the CUDA algorithms 4 and 6 for different data sizes under different numbers of variables k 2, 6, 14, 30 and 62 (from left to right)

References

[1] N.M. Adams, S.P.J. Kirby, P. Harris and D.B. Clegg, A review of parallel processing for statistical computation, *Statistics and Computing,* **6** (1996), 37 - 49.

[2] A.R. Brodtkorb, T.R. Hagen and M.L. SaeTra, Graphics processing unit (GPU) programming strategies and trends in GPU computing, *Journal of Parallel Distributed Computing,* **73** (2013), 4 - 13.

[3] M. Creel, User-Friendly Parallel Computations with Econometric Examples, *Computational Economics,* **26** (2005), 107 - 128.

[4] M. Creel and W.L. Goffe, Multi-core CPUs, Clusters, and Grid Computing: A Tutorial, *Computational Economics,* **32** (2008), 353 - 382.

[5] A.V. Dobrovidov and I.M. Ruds'Ko, Bandwidth selection in nonparametric estimator of density derivative by smoothed cross-validation method, *Automation and Remote Control,* **71** (2010), 209 - 224.

[6] A. Elgammal, R. Duraiswami and L.S. Davis, Efficient Kernel density estimation using the Fast Gauss Transform with applications to color modeling and tracking, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* **25** (2003), 1499 - 1504.

[7] J. Fernandez, M. Anguita, S. Mota, A. Canas, E. Ortigosa and F.J. Rojas, MPI toolbox for Octave, In: *Proceedings of VecPar'2004,* (2004).

[8] W. Härdle, A. Werwatz, M. Müller and S. Sperlich, Nonparametric Density Estimation, In: *Nonparametric and Semiparametric Models,* Springer Series in Statistics (2004), 39-83.

[9] D.B. Kirk and W.-m.W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach,* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[10] J. Klemel, *Smoothing of Multivariate Data: Density Estimation and Visualization,* Wiley Publishing, 1st edition, 2009.

[11] E.J. Kontoghiorghes, *Parallel Algorithms for Linear Models - Numerical Methods and Estimation Problems,* Kluwer Academic Publishers, 2000.

[12] E.J. Kontoghiorghes, *Handbook of Parallel Computing and Statistics,* Chapman & Hall/CRC, 2005.

[13] S. Lukasik, Parallel computing of kernel density estimates with MPI, In: *Proceedings of the 7th International Conference on Computational Science, Part III: ICCS 2007,* Springer-Verlag (2007), 726 - 733.

[14] P.D. Michailidis and K.G. Margaritis, Parallel computing of kernel density estimation with different multi-core programming models, In: *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'2013),* IEEE Computer Society (2013).

[15] J. Nickolls, I. Buck, M. Garland and K. Skadron, Scalable Parallel Programming with CUDA, *ACM Queue,* **6** (2008), 40 - 53.

[16] Nvidia, CUDA Compute Unified Device Architecture: Programming Guide v 3.0.

[17] Nvidia, CUDA GPU Occupancy Calculator, 2012.

[18] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone and J.C. Phillips, GPU Computing, *Proceedings of the IEEE,* **96** (2008), 879 - 899.

[19] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn and T.J. Purcell, A survey of general-purpose computation on graphics hardware, *Computer Graphics Forum,* **26** (2007), 80 - 113.

[20] J. Racine, Parallel distributed kernel estimation, *Computational Statistics and Data Analysis,* **40** (2002), 293 - 302.

[21] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.-m. W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* ACM Press (2008), 73 - 82.

[22] B. Silverman, *Density Estimation for Statistics and Data Analysis,* Chapman and Hall/CRC, 1st edition, 1986.

[23] B. Silverman, Algorithm AS 176: Kernel density estimation using the fast Fourier transform, *Applied Statistics,* **31** (1982), 93 - 99.

[24] M.P. Wand and M.C. Jones, *Kernel Smoothing,* Chapman and Hall/CRC, 1st edition, 1994.

Received: January, 2013